Precondition-based Modular Verification to Guarantee Data Race Freedom in Java Programs

Modular extension to Java RaceFinder

KyungHee Kim
Beverly A. Sanders
Neha Rungta
Eric G. Mercer

Thanks to : Tuba Yavuz-Kahveci & Peter Mehlitz

Motivation

- Modern computer architectures based on Relaxed Memory Model.
- When a program has a Data Race, Sequential Consistency is not guaranteed.
- Model Checking techniques assume Sequential consistency.

Data Race

- When two memory accesses,
- by different threads,
- on the same memory location,
- are not ordered,
- and at least one is write.

Example

- simple java program with two threads **data producer** and **data consumer**

```
class Simple
    int data=0;
    boolean done=false;
```

data producer

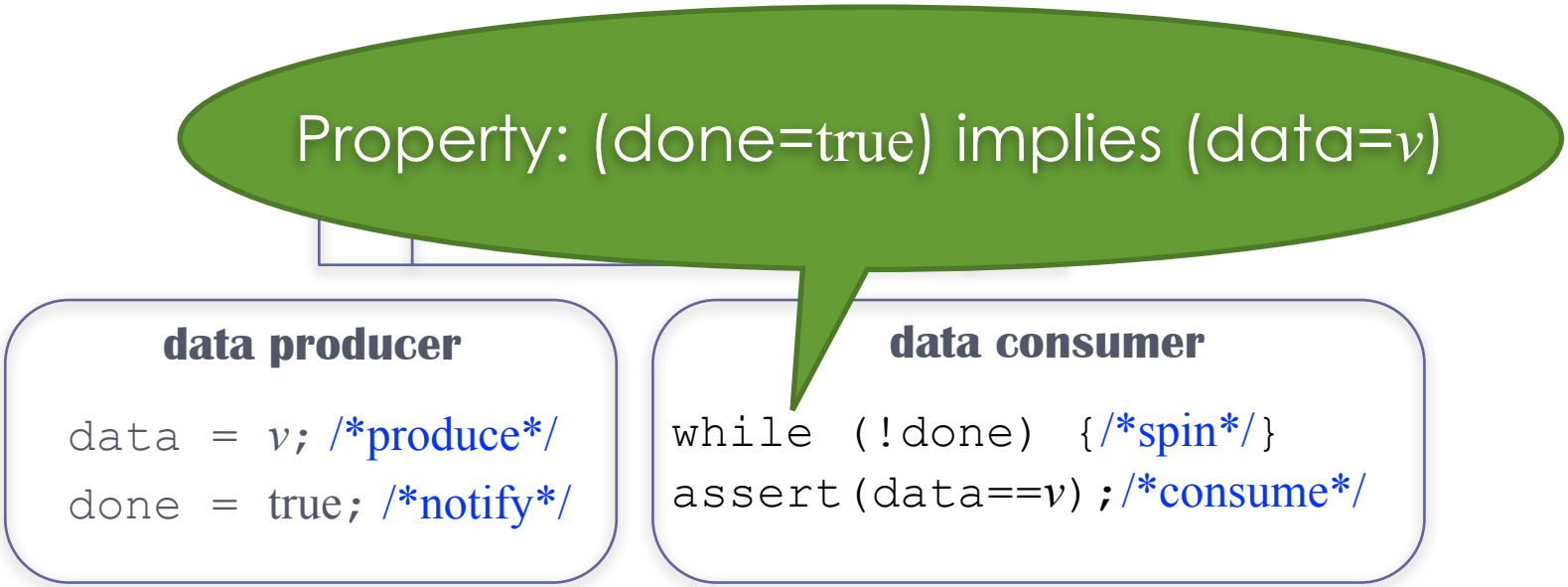
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

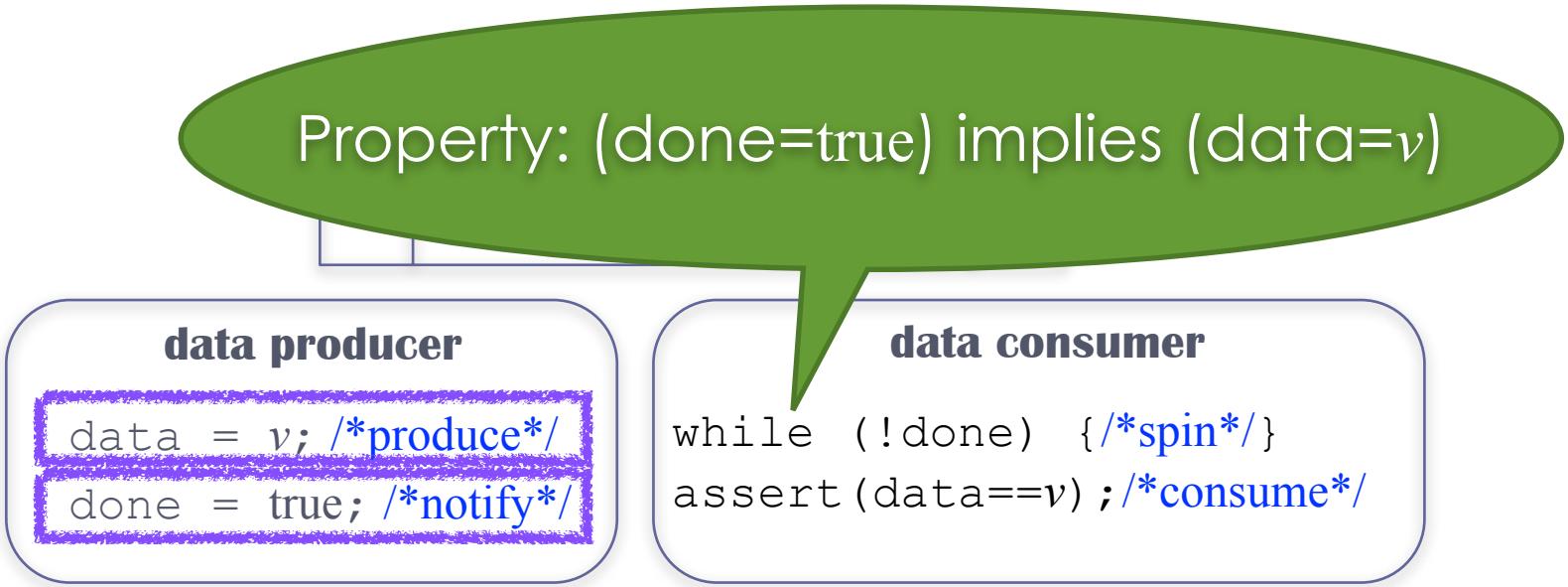
Example

- simple java program with two threads **data producer** and **data consumer**



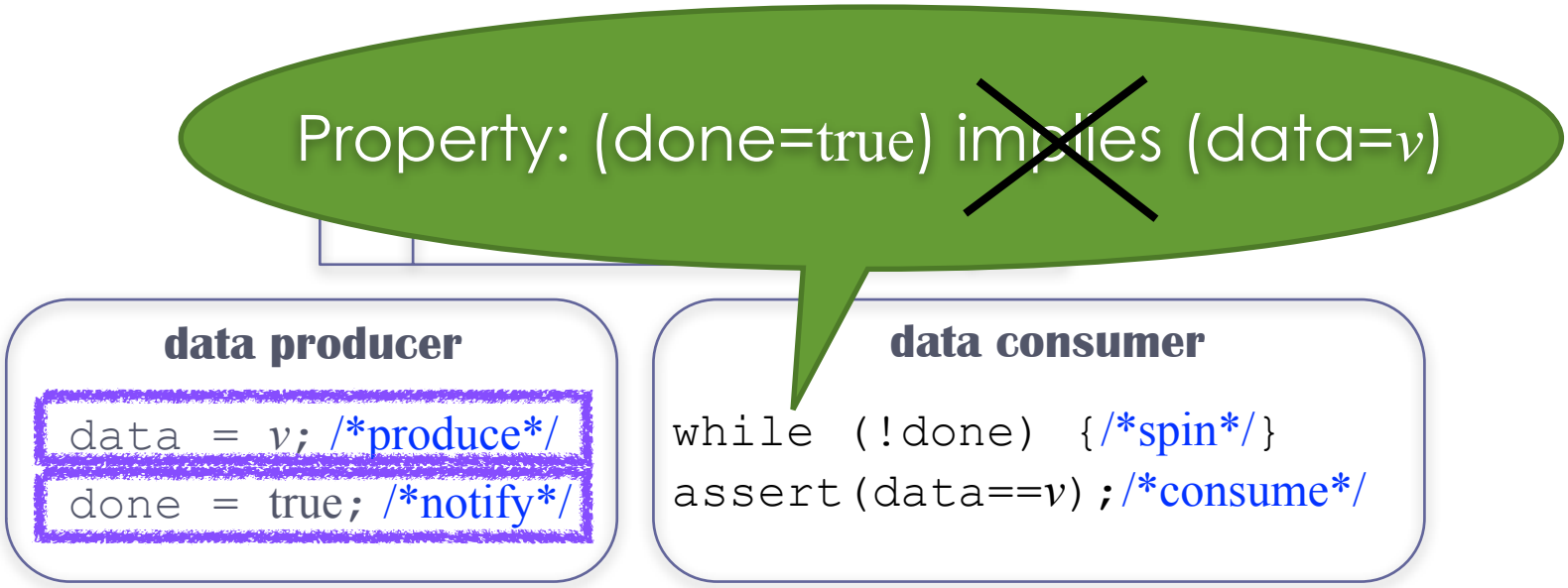
Example

- simple java program with two threads **data producer** and **data consumer**



Example

- simple java program with two threads **data producer** and **data consumer**



Example

- simple java program with two threads **data producer** and **data consumer**

```
class Simple
    int data=0;
    boolean done=false;
```

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

Example

- simple java program with two threads **data producer** and **data consumer**

```
class Simple
    int data=0;
    boolean done=false;
```

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

Solution

- Fundamental property of Java Memory Model
  - Programs whose SC executions have no races must have only SC executions.

(1) When a program is proved to be race-free, JPF can be soundly used to verify further properties such as assertion violation and deadlock.

(2) The data race freedom of a program can be proved by verifying all SC executions are free of data races.

Java RaceFinder (ASE '09)

- Java RaceFinder extends JPF to detect data races in all SC executions of a multithreaded program.
  - happens-before relation is transitive and formed by *release-acquire* pair
  - summarizing happens-before relations of events as a summary function h
  - implemented using JPF listener/visitor pattern to instrument each event
  - field factory used to workaround MJI instrumentation
  - explored additional search spaces introduced by *hb-state* (same JPF state with different summary function h)
  - tested various search algorithm including heuristic search/DFS/BFS/ random search

Java RaceFinder-Eliminator (ASE '10)

- JRF-E extends JRF to advice how to eliminate found races.
  - using counterexample path and acquiring history
  - suggest to add lock/change to volatile/move instructions/use java.util.concurrent.atomic package...
  - implemented using JPF listener/visitor to instrument each event
  - used counterexample path
- More optimizations to address state space explosion problem such as excluding threadlocal memories and standard libraries/ lazy representation of array/hb-state abstractions, etc.

Motivation (again)

- Case studies using concurrent data structure libraries
  - hard to find test driver
  - focusing on testing individual functionality of each method rather than testing their interfaces & interference
  - ➡ no easy way to test libraries
- Need to have a closed system with specific environments
- Suggestions are not applicable when they are to change libraries
- can we reduce search space by modularization?

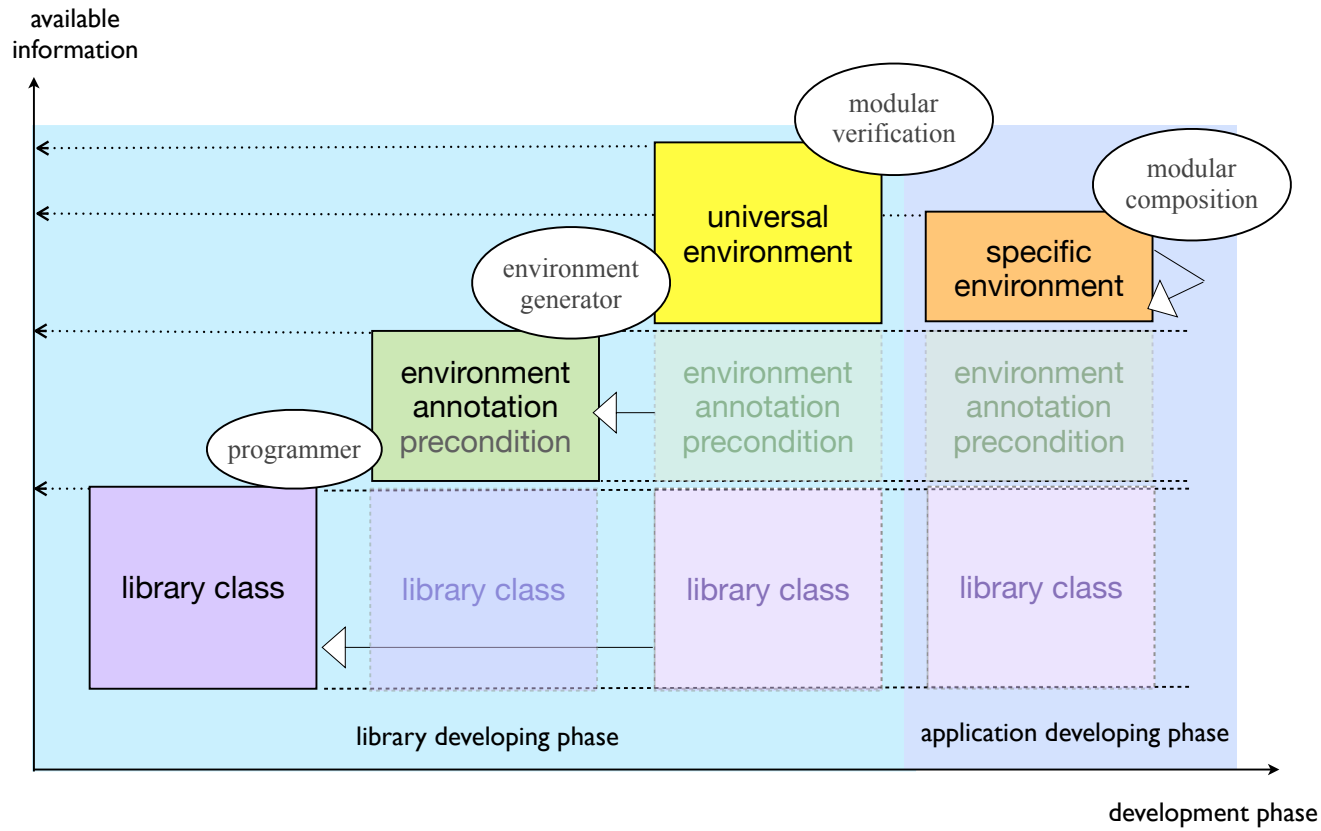
Solution (again)

- Handle application differently from libraries
- Provide a way to verify libraries against all possible use cases
- Exclude already verified libraries from race checking to achieve modular verification

Java RaceFinder-Modular (JPF '11)

- library programmer annotates his design assumptions about the preconditions necessary to ensure race freedom
- tool will generate a universal environment to prove the annotations are sufficient condition to guarantee race freedom
- tool will validate the preconditions are sufficient enough to guarantee race freedom
- uses will use the library in an application
- tool will check the preconditions for the library and JRF will check races in the codes not from the library

Operational model



Library

```
public class UnboundedQueue {
  private static final int EMPTY = Integer.MIN_VALUE;
  public final ReentrantLock enqLock, deqLock;
  Node head, tail;

  public UnboundedQueue() {
    enqLock = new ReentrantLock();
    deqLock = new ReentrantLock();
    head = new Node(EMPTY);
    tail = head;
  }

  protected class Node {
    public int value;
    volatile public Node next;
    public Node(int x) {
      value = x;
      next = null;
    }
  }
```

```
public int deq() throws EmptyException {
    int result;
    deqLock.lock();
    try {
      if (head.next == null) throw new EmptyException();
      result = head.next.value;
      head = head.next;
    } finally { deqLock.unlock(); }
    return result;
  }

  public void enq(int x) {
    if (x == EMPTY) throw new NullPointerException();
    enqLock.lock();
    try {
      Node e = new Node(x);
      tail.next = e;
      tail = e;
    } finally { enqLock.unlock(); }
  }
}
```

†

```
public int size() { /*requires enqLock, deqLock */
    int i=head==tail?0:1;
    for (Node tmp=head.next; tmp != null && tmp != tail ; tmp=tmp.next, ++i);
    return i;
}
```

Annotating methods

- Preconditions

class_annotation := **thread_bound**
method_annotation := **depth_bound** (**precondition** . . .)
precondition := (**condition_type** . . .)
condition_type := **field in** h | $lock$(**field**) | $synch$(**field**)

Annotated Library

```
@class (threads_bound=3)
public class UnboundedQueue {
  private static final int EMPTY = Integer.MIN_VALUE;
  public final ReentrantLock enqLock, deqLock;
  Node head, tail;

  public UnboundedQueue() {
    enqLock
    deqLock
    head = new Node(EMPTY);
    tail = head;
  }

  protected class Node {
    public int value;
    volatile public Node next;
    public Node(int x) {
      value = x;
      next = null;
    }
  }
```

```
@method (depth_bound=1) // default for constructor
```

```
@method (depth_bound=2)
public int deq() throws EmptyException {
@precondition (h="CURRENT_THREAD WITH THIS")
// default safe publication condition
      if (head.next == null) throw new EmptyException();
      result = head.next.value;
      head = head.next;
      finally { deqLock.unlock(); }
    return result;
```

```
@method (depth_bound=2)
public void enq(int x) {
@precondition (h="CURRENT_THREAD WITH THIS")
// default safe publication condition
      Node e = new Node(x);
      tail.next = e;
      tail = e;
    } finally { enqLock.unlock(); }
  }
```

```
@method (depth_bound=2)
@recondition (lock="enqLock", lock="deqLock")
```

```
public int size() { /*requires enq
    int i=head==tail?0:1;
    for (Node tmp=head.next; tmp != null
    return i;
}
```

```
@precondition (h="CURRENT_THREAD WITH THIS")
// default safe publication condition
```

Universal Environment Generation

- use data choice to cover method sequences
- use symbolic parameter to cover all possible inputs
- generate lock/synchronization specified in preconditions

Generated universal environment

```
public class UnboundedQueueVerify {
   public static void main(String[] args)
   {  UnboundedQueueVerify().doTest(); }

   UnboundedQueue obj;


   void doTest()  {
      for ( int i=0 ; i < 1 ; ++i)
          new Group1Thread().start();
      for ( int i=0 ; i < 3 ; ++i)
          new Group2Thread().start();
   }

class Group1Thread extends Thread {
      public void run() {
         for ( int i=0 ; i < 1 ; ++i) {
            int c = gov.nasa.jpf.jvm.Verify.getInt(1, 1);
            if ( c == 1 ) obj = new UnboundedQueue();
         }
      }
}
```

```
class Group2Thread extends Thread {
   public void run()
   {
      while(obj==null);
      for ( int i=0 ; i < 6 ; ++i ) {
         int c = gov.nasa.jpf.jvm.Verify.getInt(1, 3);
         if ( c == 1 ) {
             obj.deqLock.lock(); obj.enqLock.lock();
             try { obj.size(); }
             finally {
                 obj.enqLock.unlock();
                 obj.deqLock.unlock();
             }
         }
         else if ( c == 2 ) {
             @Symbolic("true")
             int sym1=0;
             obj.enq(sym1);
         }
         else if ( c == 3 ) {
             try { obj.deq(); }
             catch (EmptyException e) {}
         }
      }
   }
}
```

Verification of the race freedom of the module

- JRF will detect any remaining races
- ignore executions where constraints violated
- adjust summary function h when h precondition is violated

Verification of the application

- JRF will detect races in application code
- JRF-modular will detect precondition and constraint violation

Example Application

```
public class FairMessage {
      static final int PER_THREAD=2, NUM_THREAD=2;
      UnboundedQueue queue = new UnboundedQueue();
      DisBarrier bar = new UnboundedQueue();

      public static void main(String[] args) {
          new FairMessage().run();
       }
      private void run()
      {
          for (int i=0 ; i < NUM_THREAD ; ++i) {
                new EnqThread(i).start();
                new DeqThread().start();
          }
          System.out.println("queue size="+quene.size());
      }
```

```
class EnqThread extends Thread {
      int  id;
      EnqThread(int i) { id = i; }
      public void run() {
        for (int i = 0; i < PER_THREAD; i++) {
          queue.enq(id + i);
        }
      }
}
class DeqThread extends Thread {
      public void run() {
          for (int i = 0; i < PER_THREAD; i++) {
             try { queue.deq(); bar.await(); }
             catch (EmptyException ex) {}
          }
      }
 }
```

JRF result for FairMessage

```
================================================
JRF results
================================================ data race #1
edu.ufl.cise.jrf.util.HBDataRaceException
         at THREAD      (java.lang.Thread@ from null)
         to MEMORY      (jrfm.UnboundedQueue@.tail
                          from "volatile UnboundedQueue queue = new UnboundedQueue();"
                          at jrfm/FairMessage.java:10 in (<init>))
         in INSTRUCTION (getfield)
         of SOURCE      ("for (Node tmp=head.next; tmp!=null &&
                                      tmp!=tail ; tmp=tmp.next, ++i);"
                          at jrfm/UnboundedQueue.java:72)
. . .
```

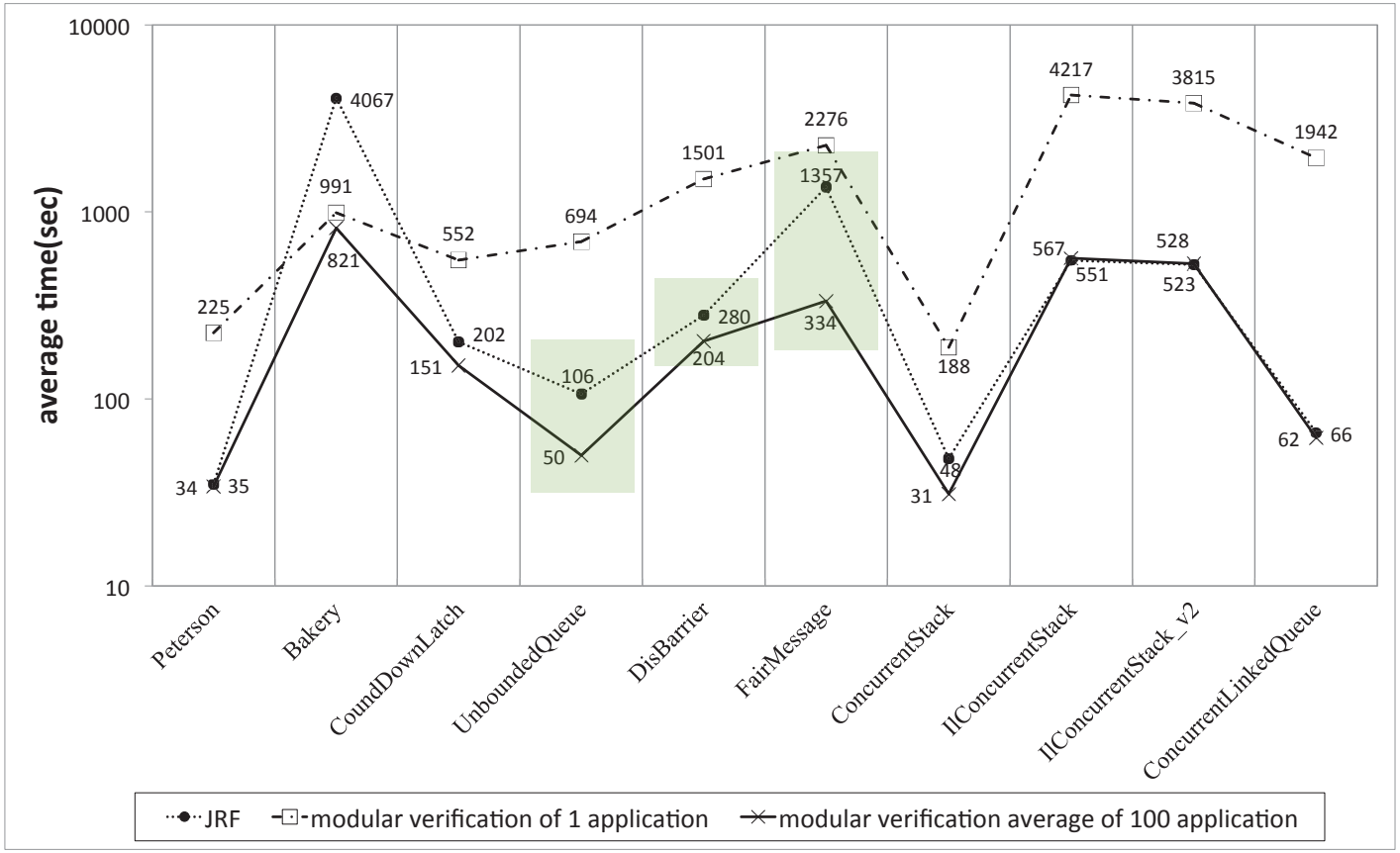
JRF-E result for FairMessage

```
================================================
JRF-E results
_____________________________________________________ analyze counter example
data race source statement : "putfield" at jrfm/UnboundedQueue.java:58  :
          "tail = e;"
          by thread 1
data race manifest statement : "getfield" at jrfm/UnboundedQueue.java:72:
          "for (Node tmp=head.next; tmp!=null &&
                        tmp!=tail ; tmp=tmp.next, ++i);"
          by thread 0

Change the field "jrfm.UnboundedQueue@.tail
         from "volatile UnboundedQueue queue = new UnboundedQueue();"
         at jrfm/FairMessage.java:10 in (<init>)" to volatile.

    Lock "java.util.concurrent.locks.ReentrantLock@
             from "enqLock = new ReentrantLock();"
             at jrfm/UnboundedQueue.java:25 in (<init>)"
             before accessing (jrfm.UnboundedQueue@.tail)
. . .
```

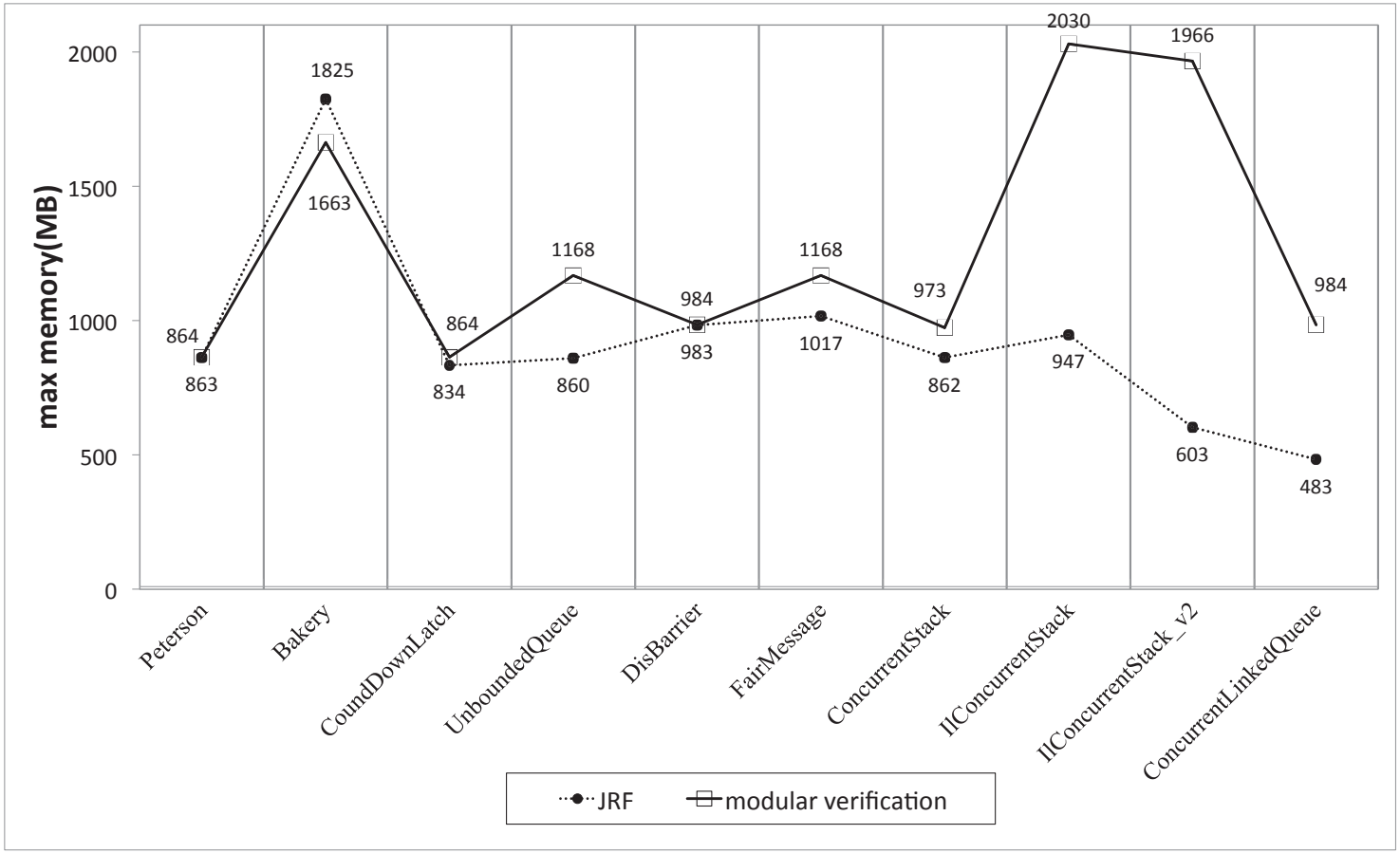
JRF-Modular result for FairMessage

```
================================================
JRFM-ComposeModule results
================================================ precondition violation #0
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "System.out.println("queue size = "+queue.size());"
     in "jrfm.FairMessage.run(FairMessage.java:23)"
================================================ precondition violation #1
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "assert (queue.size() == 0);"
     in "jrfm.FairMessage.run(FairMessage.java:17)"
. . .
```

Experimental Result (time)



Experimental Result (max memory)



Conclusion

- Preconditions are a great way to share the library developer's design choices with its users.
- Universal environment benefits the library developer to debug concurrency issues.
- Given that a library is verified once and the result can be applied to many times, JRF-modular extension outperforms JRF and JRF-E in spatial and temporal performance.
- Moreover, this advice in more appropriate in that it only suggests to fix application rather than the library code.

Future Work

- Extend to include race-free invariant
- How to overcome the limitation of constraints
- How to address the space explosion problem with symbolic parameters and data choices

Question?

Thank you